Procedia Computer Science

Volume 80, 2016, Pages 484–495

ICCS 2016. The International Conference on Computational Science

Competing Energy Lookup Algorithms in Monte Carlo Neutron Transport Calculations and Their Optimization on CPU and Intel MIC Architectures

Yunsong Wang[1], Emeric Brun[2], Fausto Malvagi[2], and Christophe Calvin[3]

[1] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ, Univ. Paris-Saclay
yunsong.wang@cea.fr
[2] CEA/DEN/DANS/DM2S/SERMA
[3] CEA/DRF/D3P
CEA Saclay, F-91191, Gif-sur-Yvette, France

Abstract

The Monte Carlo method is a common and accurate way to model neutron transport with minimal approximations. However, such method is rather time-consuming due to its slow convergence rate. More specifically, the energy lookup process for cross sections can take up to 80% of overall computing time and therefore becomes an important performance hotspot. Several optimization solutions have been already proposed: unionized grid, hashing and fractional cascading methods. In this paper we revisit those algorithms for both CPU and manycore (Intel MIC) architectures and introduce vectorized versions. Tests are performed with the PATMOS Monte Carlo prototype, and algorithms are evaluated and compared in terms of time performance and memory usage. Results show that significant speedup can be achieved over the conventional binary search on both CPU and Intel MIC. Further optimization with vectorization instructions has been proved very efficient on Intel MIC architecture due to its 512-bit Vector Processing Unit (VPU); on CPU this improvement is limited by the smaller VPU width.

Keywords: Monte Carlo, neutron transport, cross section, table lookup, Intel MIC, vectorization

1 Introduction

Monte Carlo (MC) transport simulations are widely used in the nuclear community to perform reference calculations. This method simulates the physics by following a neutron in its travels inside a system from birth to absorption or leakage. This random walk is governed by interaction probabilities described by microscopic cross sections. Macroscopic quantities like neutron densities can be estimated from large samples of histories, which makes the MC method more computationally expensive than other approaches. The advantage of the MC method is that

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.327

since the individual histories are independent, it is an ideal candidate for parallel computing. On the other hand, each history is different and there is little evidence of natural vectorization.

Several difficulties have been identified when porting MC codes to modern architectures: hybrid parallelism, memory sharing, cache misses, limited use of vector processing units. Modern computing accelerators evolve constantly and have distinct strategies which thus require specific optimization efforts.

In order to explore both the challenges and the benefits brought by emerging accelerators for the MC codes, a new MC neutron transport prototype called PATMOS (PArticle Transport Monte Carlo Object-oriented System) is currently under development at CEA [1]. PATMOS is lightweight and portable, but also complex enough to represent a real simulation. Its goal is to test competing algorithms on a variety of hardware so as to explore their computing potentials and get the best compromise between performance and sustainability of the code. This prototype is entirely written in C++ standard 11/14 [2]. PATMOS implements a hybrid parallelism based on MPI and OpenMP or standard C++ threads. In a multi-threaded environment, the total number of particles to be simulated is evenly dispatched to all the available threads.

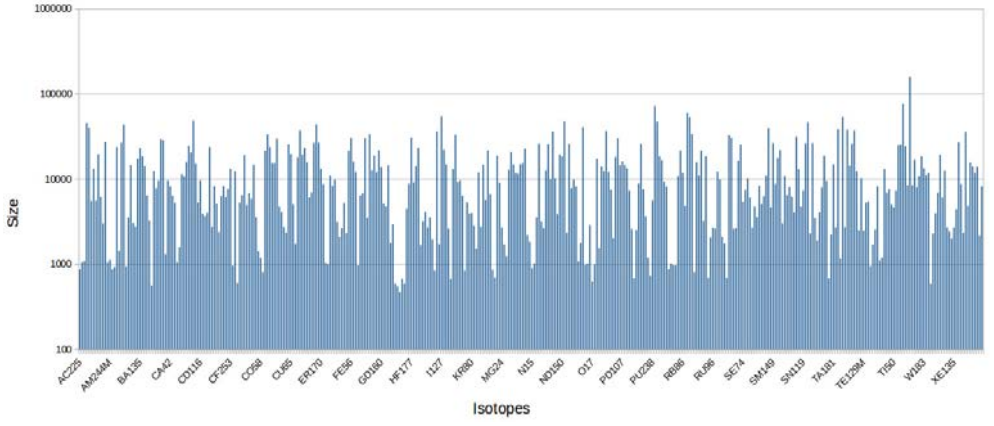

Figure 1: Isotopic energy table lengths for 390 isotopes at $T = 300$K. The minimum is for $^{3}\mathtt{H}$, which has only 469 energy points, and the largest is for $^{238}\mathtt{U}$ with 156,976 points. The average length over all isotopes is around 12,000.

In MC transport calculations, the cross sections represent the interaction probabilities of the particle with the underlying medium. Isotopic cross sections, which depend on the kinetic energy of the incident particle, are stored in tables of up to 150,000 couples (E_i, σ_i) meant for linear-linear interpolation (Figure 1). These isotopic tables typically have between 10^2 and 10^5 values, depending on the variations of the cross-sections, number of resonances, threshold reactions, etc [3].

Every time a particle with energy E suffers a collision or crosses a material interface, the material total cross section $\Sigma(E)$ needs to be recalculated as the sum of all isotopic cross sections $\sigma_i(E)$ times the isotopic concentrations N_i. That is $\Sigma(E) = \sum_i N_i \sigma_i(E)$, where the isotopic cross sections are computed as:

$$\sigma_i(E) = \sigma_i(E_k) + \frac{E - E_k}{E_{k+1} - E_k}[\sigma_i(E_{k+1}) - \sigma_i(E_k)]$$

where E_k and E_{k+1} are the nearest lower and upper energy points in the isotopic energy grid which bound the random energy E.

It has been pointed out that for a MC code, this process may take up to 80% of overall computing time [3, 4, 5]. Thus any efforts to accelerate this energy lookup process will be of great interest to the MC community. Previous studies focused on this issue have already proposed several optimization solutions: unionized energy grid [4], hashing method at material level [3] and fractional cascading method [5]. One recent work [6] has compared a number of optimizations on unionized methods and hashing methods with the MC code OpenMC [7], and their results show that overall code speedup factors of 1.2-1.5× can be obtained compared to the conventional binary search. But the work did not address the issue of evolving architectures and especially manycore ones.

This paper describes the use of PATMOS to analyze the performance of a large collection of energy lookup methods on both CPU and MIC architectures. In order to take advantage of the VPU in modern processors and in particular in MIC, we have tested a vectorized linear search scheme and a new SIMD-based energy lookup method. Section 2 presents the algorithms chosen for this study. Section 3 describes the vectorizations and other optimizations developed in this work. Section 4 is devoted to numerical results and algorithm evaluation in terms of performance, scalability and memory footprint. Section 5 presents some concluding remarks.

2 Competing Energy Lookup Algorithms

Our goal is to test algorithms that perform well on both CPU and manycore architectures. The following issues are of primary importance when considering potential algorithm performance:

- `Degree of parallelism`: The current `Knights Corner` MIC coprocessor consists of up to 61 in-order cores running at 1GHz and each core supports 4 hardware threads. Since one of these 61 cores is reserved for the system, generally 240 threads can be used for the program. Note that every individual core is much less powerful than that of CPU, and code with low degree of parallelism is not suited for MIC.

- `Vectorization`: MIC architecture is more than a multi-core system due to its dual-issue pipeline and the 512-bit wide VPU. It has been pointed out that fully utilizing the VPU is critical for best coprocessor performance [8]. Algorithms with little vectorization potential are not suited for manycore implementation.

- `Memory footprint`: Compared to the CPU, memory size in the current manycore architecture is relatively small and limited to 16GB. Algorithms with big memory requirements must be redesigned.

Because of the considerable variations between different isotopes (Figure 1), finding a generic energy lookup solution well-suited for all isotopes is difficult. Traditionally, a simple binary search is employed to perform energy indexing in the original cross section tables. However, retrieving data in large tables with the bouncing binary search results in high cache misses (65% at last level cache according to Tramm [9]) and therefore degrades efficiency. Moreover, this algorithm offers very little opportunity for vectorization.

A number of methods have been proposed in the past as alternatives to the binary search:

- `Hashing`: each material's whole energy range is divided up into `N` equal intervals, and for every individual isotope inside the material an extra table is established to store isotopic bounding indexes of each interval [3]. The new search intervals are thus largely narrowed with respect to the original range and can be reached by a single float division. The hashing can be performed on a linear or logarithmic scale; the search inside each interval

can be performed by a binary or linear search. In the original paper [3], a logarithmic hashing was chosen with $N \simeq 8000$ as the best compromise between performance and memory usage. Another variant is to perform the hashing at the isotope level.

- `Unionized grid`: A global unionized table gathers all possible energy points in the simulation and a second table provides their corresponding indexes in each isotope energy grid [4]. Every time an energy lookup is performed, only one search is required in the unionized grid and the isotope indexes are directly provided by the secondary index table. Timing results show that this method has a significant speedup over the conventional binary search but can require up to 36× more memory space [5].

- `Fractional cascading`: This is a technique to speed up search operations for the same value in a series of related data sets [5]. The basic idea is to build a unified grid for the first and second isotopes, then for the second and third, etc. When using this mapping technique, once we find the energy index in the first energy grid, all the following indexes can be read directly from the extra index tables without further computations. Compared to the global unionized method, the fractional cascading technique greatly reduces memory usage.

Another possibility that we propose for its vectorization and cache-locality properties is the following variation on N-ary tree:

- `SIMD-Based Search Method` This method is similar to hashing, but the energy intervals are not built on equal energy intervals but on equal number of energy grid points. The idea is the following, where we use `KNC` vectorization intrinsics:

  1. For each isotope, divide the energy grid `E` into 32 segments with range $r = \frac{grid\ size}{32}$
  2. Create an extra indexing list `I` containing $r \times j, (j = 0, 1, 2...31)$ and insert the index of last element in the energy grid at the end of the list
  3. Store 32 variables `E[I[j]]` (`j = 0,1,2...31`) into 4 `__m512d` vector variables

  Such method creates only a few hundreds bytes of extra mapping information for each isotope. The procedure for indexing an energy value is represented as following steps:

  1. Load 8 copies of `key` value to `vec_key` (with instruction **vmovapd**)
  2. Compare `vec_key` with 4 `__m512d` vector variables, store 4 comparison results in 4 8-bit variables: `cmp0`, `cmp1`, `cmp2`, `cmp3` (**vcmppd**)
  3. Pack 4 results into one 32-bit unsigned variable: `res` = `cmp0` + (`cmp1`$\ll$8) + (`cmp2` + (`cmp3`$\ll$8))$\ll$16
  4. Find the final index with a binary search in the interval between `E[I[res-1]]` and `E[I[res]]`

All the preceding algorithms have been implemented in PATMOS and re-evaluated onto MIC architecture.

3 SIMD Optimization for Linear Search

The conventional binary search in MC codes is not adapted to vectorization since the energy grid vector is (quasi-) randomly accessed. With the help of hashing, the search interval is dramatically narrowed to a small range with often only a few elements, where a simple linear search may fully profit from cache effects and become more efficient than the binary search. Moreover, the linear search performed on a continuous memory space can be vectorized in order to take advantage of the powerful VPU on MIC. One former study [10] provides a comprehensive view of optimizing linear and binary search with SSE2 instructions. In this section, our optimizations follow the ideas of this work and focus on a small sorted array of `double` variables with Advanced Vector Extensions (`AVX`) and MIC `KNC` intrinsics.

```
index = 0;
Load 8 copies of key value to vec_key (vmovapd);
for do
    Load 8 (from array[index] to array[index+7]) array elements to vals (vmovapd);
    Compare vals with vec_key (vcmppd), store the result in res;
    if res ≠ 0 then
        break;
    end
    index += 8;
end
Count trailing zero bits of res (tzcnt), store the result in offset;
return index + offset;
```

Algorithm 1: Basic vectorized linear search on MIC.

```
index = 0;
Load 8 copies of key value to vec_key (vmovapd);
for do
    Load 8 (from array[index] to array[index+7]) array elements to vals0 (vmovapd);
    Load another 8 (array[index+8] to array[index+15]) elements to vals1 (vmovapd);
    Compare vals with vec_key (vcmppd), store the result in res0;
    Compare vals with vec_key (vcmppd), store the result in res1;
    res = res0 + (res1≪8);
    if res ≠ 0 then
        break;
    end
    index += 2×8;
end
Count trailing zero bits of res (tzcnt), store the result in offset;
return index + offset;
```

Algorithm 2: Branchless vectorized linear search on MIC.

The basic SIMD algorithm for linear search on MIC architecture is represented in Algorithm 1. The MIC 512-bit VPU allows 8 simultaneous `double` operations. Our implementation relies on SIMD intrinsics instructions. For each iteration in the loop, 8 elements in the array pointed by `index` are compared with the `key` value. The comparison result is stored in a 8-bit `char`

variable. Once this result is no longer zero, we determine the `offset` by counting trailing zero bits of this result. The final result is the accumulated `index` plus `offset`. In the basic algorithm, there is one conditional branch per 8 array elements, such proportion can still be reduced by unrolling the loop (Algorithm 2). With the branchless method, each iteration makes 16 comparisons and the final `offset` is packed from 2 8-bit variables with a simple bit-set operation.

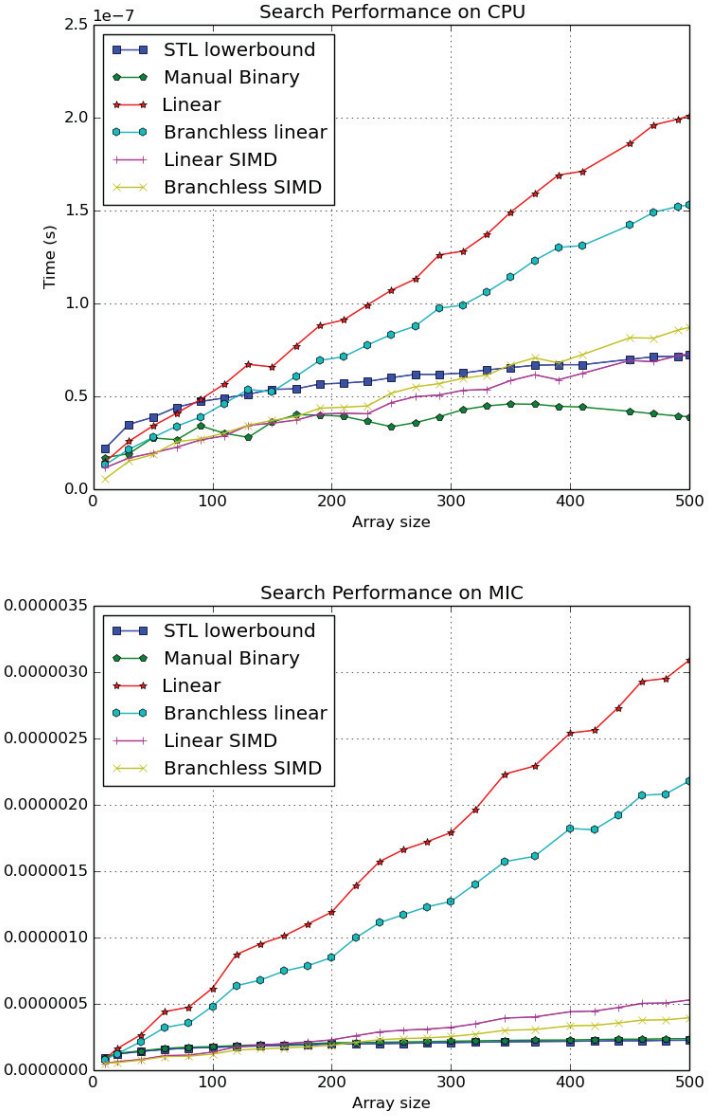

Figure 2: Performance of several versions of binary and linear search as a function of array size.

Algorithms of vectorized linear search on CPU is quite similar to MIC, except that only 4 `double` variables (with AVX instruction set) can be handled simultaneously. Besides, com-

parison results with CPU intrinsics are still in the VPU register. Instructions like **packssdw**, **packsswb** and **vmovmskpd** are needed to pack multiple results and move them into a general purpose register.

The experimental environment for all tests presented in this paper is comprised of a `Sandy Bridge E5-2670` bi 8-core processor at 2.6GHz and a 61-core 1.2GHz `KNC` coprocessor. Figure 2 represents timing results of different search algorithms performed on these two architectures respectively. Serial tests are performed on varying array size from 10 to 500 in order to record the pure search performance. SIMD optimization brings significant speedup on MIC. The vectorized branchless linear search may bring a factor of 10× improvement over the basic linear search. On CPU, however, it performs less efficiently than the basic vectorized linear search. For any array less than 200 elements, the optimized linear search turns out to be always more efficient than the binary search on MIC. Similar conclusions can be drawn on CPU as well. Results indicate that the threshold array size for choosing between the linear or the binary search is 200. Under this value, the linear search is more efficient.

4 Results and Analysis

In this section we present our optimization work for several energy lookup algorithms and their test results performed in full transport simulations with PATMOS. Our generic test case (called `PointKernel`) is the neutron simulation of a slowing down problem from a 2 MeV source in an infinite medium composed of all the 390 isotopes of the nuclear data library; the main components of the mixture are ^{1}`H` and ^{238}`U` in order to have a classical Pressurized Water Reactor (PWR) spectrum, the other isotopes intervening as trace elements. This simulation is representative of PWR burn-up computations.

4.1 Efficient Hashing Strategies

The efficiency of hash-based methods depends on several factors. Unit tests are carried out respectively for each aspect in order to find the optimal solution.

The hashing size `N` determines the number of energy points in the hash bins. Greater `N` number means more bins and therefore fewer elements in each bin. In theory, larger hashing size requires more memory space but performs better. It has been mentioned [3] that dividing the whole energy range into $N \simeq 8000$ segments is a reasonable compromise between performance and memory usage.

Our first test case `PointKernelU238` is a variation of `PointKernel`, where the only isotope present is ^{238}`U`. We have carried out the test varying the hashing size `N` from 200 to 32,000, and found that a larger hashing size always provides better search efficiency. But when we switched to the `PointKernel` test, where all 390 isotopes are present, $N \simeq 500$ is observed to be the most efficient value with a gain of around 7% over the original optimal $N \simeq 8000$. This may be due to the fact that the average energy grid size is around 12,000 (see Figure1) which is not much bigger than 8,000.

In the hashing method at isotope level, we can specify a different hashing strategy for each isotope, based on its own energy grid properties. Following this idea, we implemented a hashing method with $N = 500$ for energy grids with less than 30,000 elements and $N = 10000$ for the rest. Results show that such adapted hashing size provides a 5% speedup.

The energy point distributions are very irregular, due to resonances where the cross sections change of several order of magnitude in intervals of just 1 eV. Hashing the 390 isotopes with $N = 500$ for example, the average element in each bin is around 21. But most hash bins

have no more than 5 elements, while in the resonance region isotopes like ^{239}PU, ^{238}U may have bins with up to 4000 energy points. Such unbalanced distribution leads to performance degradation. In order to investigate this issue, we measured average search time in each hash bin. The results confirm that larger hash bins take longer during indexing, but timing differences between isotopes are not very important (see Table 1). For example, the largest ^{238}U energy grid is 246× more voluminous than the smallest ^{3}H, but indexing energy points using linear search in a ^{238}U hash bin is on average only 1.5× longer than in ^{3}H.

Isotope	Energy grid size	Average bin size	Time (s)
^{3}H	469	1	2.473e-07
^{127}I	50,759	101.52	2.997e-07
^{239}PU	53,284	106.57	3.039e-07
^{235}U	53,936	107.87	3.058e-07
^{238}U	122,567	245.13	3.756e-07
Average:	10,761	21.52	2.49e-07

Table 1: Search performance in the hash bin (with N=500).

The hashing method revisited by Brown [3] is based on a logarithmic scale. It is not obvious that such hashing organization is optimal for all cases. Therefore, Walsh et al. [6] proposed to establish the hash function simply on a linear scale. In the benchmark `PointKernel`, we observed that linear scale takes up to 2.2× more time than the logarithmic scale and performs even slightly slower than the conventional binary search.

A detailed analysis of energy point distributions show that while a logarithmic scale is the most appropriate to have balanced hash bins outside the resonance region, when resonances are involved a linear scale may be more effective for bin balance. We thus implemented a mixed hashing, using linear scale between 1eV and the end of the resolved resonance region and a log scale otherwise. Timing results performed on the `PointKernelU238` test case show that this mixed hashing method brings no acceleration on either CPU or MIC and it performs a little slower (no more than 10%) than the pure logarithmic method. Such performance degradation comes from the thread divergence when determining in which hashing region the lookup is to be carried on. Further optimization by varying hash size and reorganizing hashing region could still be worth exploring.

4.2 Optimized Access Pattern for Unionized Method

PATMOS employs the double indexing unionized method introduced by Leppänen [4], which utilizes a pointer to access isotopic energy grids from the global unionized table. In our first implementation, an indexing table is constructed for each isotope; we then combine all the list into one 2D indexing table. For the benchmark `PointKernel`, for example, we have an indexing table with 390×3,574,598 elements.

Profiling results show that access to the indexing table is one of the performance hotspots for the unionized method, because the indexes for each isotope belong to different arrays. A better data structure can be accomplished by simply transposing the indexing table to 3,574,598×390 form: for each energy points in the unionized grid, the corresponding indexes of all 390 isotopes are contiguous in the memory space and can be loaded once for all. Table 2 shows that such an optimization can bring a speedup between 20% and 50% for the simulation, but with worsening performance as the number of threads increases.

	Serial	16 threads
Original	131.39 s	12.22 s
Optimized	85.06 s	10.23 s
Speedup	1.54×	1.20×

(a) Results on CPU.

	60 threads	240 threads
Original	60.92 s	25.18 s
Optimized	39.56 s	20.63 s
Speedup	1.54×	1.21×

(b) Results on MIC.

Table 2: Speedup of optimized unionized method for the `PointKernel` test case.

4.3 Reordered Fractional Cascading Grid

The fractional cascading method builds indirection indexes for the isotopes two by two. Since every isotope has an energy grid length largely differing from each other and we always perform a search process only in the first mapping table, the order in which the energy grids maps are generated may have a significant effect on searching performance.

	Serial	16 threads
Alphabetical	135.09 s	14.94 s
Min-Max	124.21 s	14.21 s
Speedup	1.09×	1.05×

(a) Results on CPU.

	60 threads	240 threads
Alphabetical	74.94 s	28.45 s
Min-Max	65.65 s	26.19 s
Speedup	1.14×	1.09×

(b) Results on MIC.

Table 3: Effects of isotope ordering when constructing fractional cascading maps for the `PointKernel` test case.

We thus tested several ordered approaches: a) random; b) alphabetical; c) from longer to shorter grid; d) from shorter to longer grid. The first three approaches gave similar results, while the ordering of grids from shortest to longest gave a performance improvement of 10% (see Table 3).

4.4 Performance and Scalability

	CPU (16 threads)			MIC (60 threads)			MIC (240 threads)		
	Time (s)	Speedup	Memory (MB)	Time (s)	Speedup	Memory (MB)	Time (s)	Speedup	Memory (GB)
Binary	32.90	1.00×	514	107.28	1.00×	970	35.76	1.00×	2.88
Cascading	14.21	2.32×	640	65.65	1.63×	1126	26.19	1.37×	2.9
HashIsotope	15.59	2.11×	527	69.81	1.54×	1030	25.86	1.38×	2.89
HashMaterial	14.05	2.34×	526	68.51	1.57×	1027	24.33	1.47×	2.89
SIMD	-	-	-	84.96	1.26×	982	31.52	1.13×	2.88
Unionized	10.23	3.22×	5862	39.56	2.86×	6348	20.63	1.73×	8.0

Table 4: Performance and memory usage for energy lookup algorithms for the `PointKernel` test case.

Figure 3 shows timing performance for different algorithms in the test case `PointKernel`. Binary search is our reference method and also the slowest. Unionized grid is the most efficient time-wise, but at the cost of a dramatical increase in memory foot-print (times 11× on CPU according to Table 4). The two variants of hashing methods and fractional cascading have nearly the same efficiency on both CPU and MIC. A disappointing fact is that with no algorithm we have been able to run faster on MIC than on CPU. Even with the help of vectorization,

algorithms run with 240 threads on MIC are slower than (and sometimes almost twice as slow as) with 16 threads on CPU. This is partly due to the fact that the code snippets which have been vectorized are not enough computationally intensive. It should be noted that in the present prototype certain data structures related to isotopic exiting distributions is still replicated for each working thread, requiring about 10MB of supplementary memory space per thread. While when working with CPU this is hardly noticeable, in MIC architecture the added cost in memory footprint is far from negligible (see Table 4). Further work on PATMOS objects is necessary in order to reduce duplication to the strict minimum.

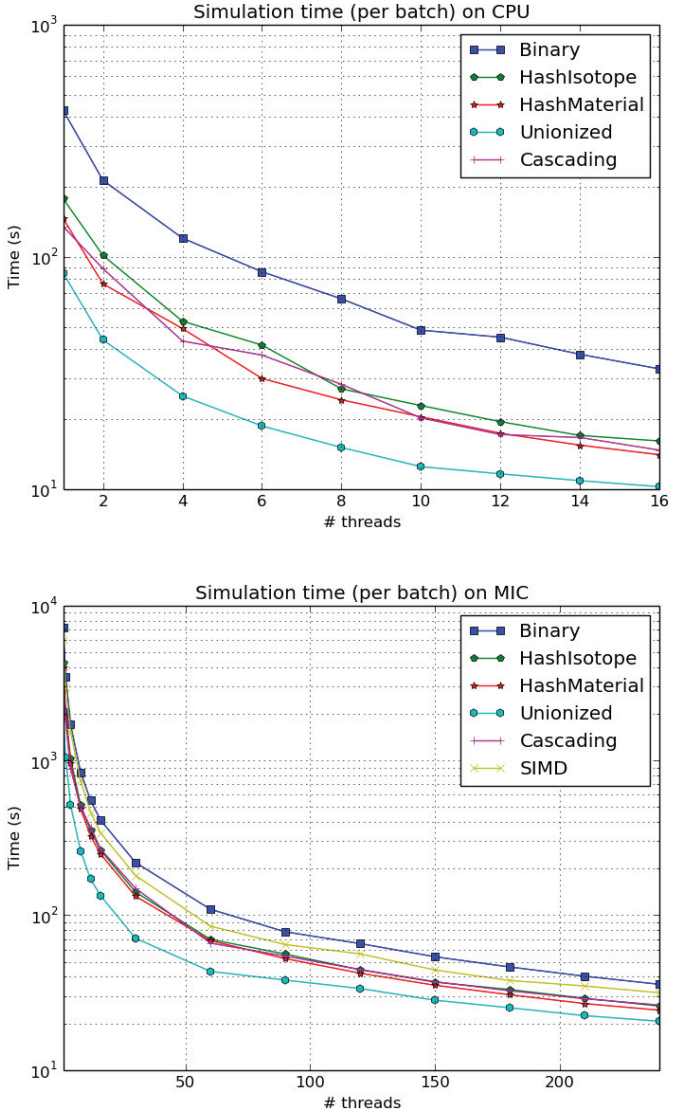

Figure 3: Performance of different energy lookup algorithms for the `PointKernel` test case.

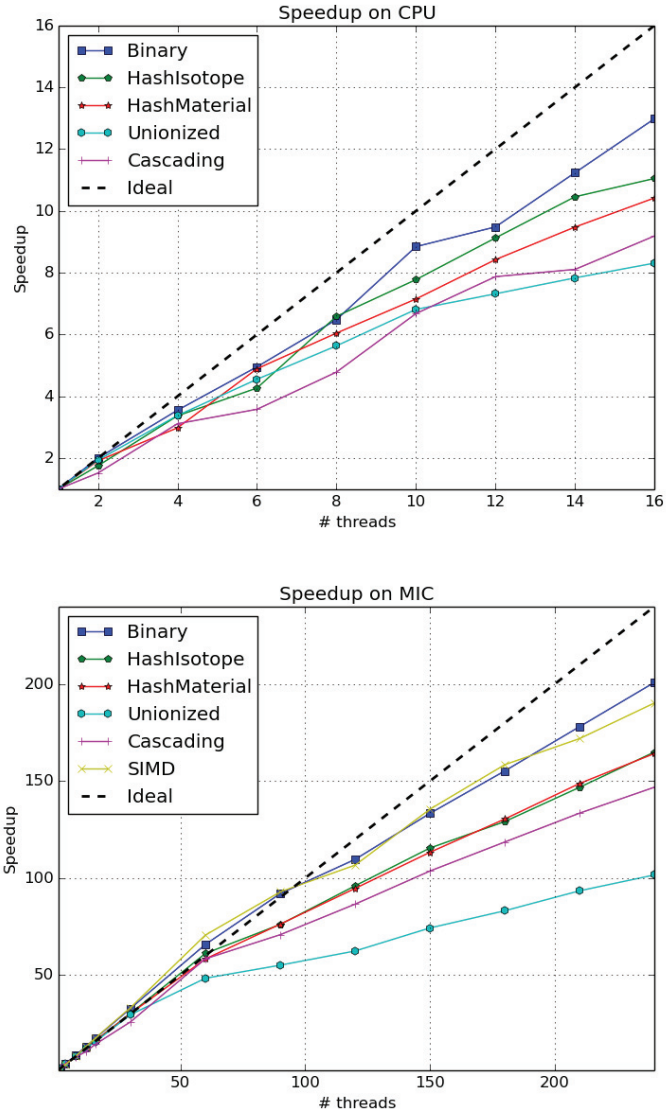

Figure 4: Speedup for energy lookup algorithms for the `PointKernel` test case.

As for the algorithm scalability (Figure 4), we note that algorithm efficiency on MIC with 60 threads is much better than the efficiency on CPU with 16 threads. Though hyper-threading with 4 threads per core results in better performance on MIC, the algorithm efficiency degrades as well. Besides, memory-bound algorithms like the unionized grid or the fractional cascading lose efficiency when augmenting the number of threads: the fractional cascading has a speedup of 10% over the material hashing with a single thread but performs always less efficiently by utilizing all threads on both CPU and MIC. Finally, we arrive at the conclusion that algorithms with better performance have a strong tendency to have worse scalability. As a result, it

seems that the logarithmic material hashing method is the best balance between performance, scalability and memory footprint.

5 Conclusions

In this paper, we have described different energy lookup algorithms implemented in PATMOS and their optimization.

In unit-tests outside the prototype, we were able to show that the vectorized linear search provides substantial gains on both CPU and MIC architecture. For an array of `double` variables, the SIMD-optimized linear search would be more efficient than the binary search when array size is less than 200. The speedup provided by MIC 512-bit VPU is relatively more significant than that of CPU due to the larger VPU width. By using such optimized search scheme in the simulation, visible additional speedup can be observed over existing solutions.

A new SIMD-based lookup algorithm has been proposed and tested for MIC architecture. It produces a 13% speedup over the conventional binary search with negligible increase in the memory footprint.

Regarding the performance of all tested algorithms, the unionized method is the fastest on both CPU and MIC architecture but at the cost of an order of magnitude increase of the memory footprint. There are no big timing differences between the two hashing methods and the fractional cascading in our test case. On the other hand, the time-consuming binary search has the best scalability among all the methods. In short, algorithms with better performance have worse scalability.

Finally, it can be concluded that the logarithmic material hashing method is a good compromise between performance and memory usage on both CPU and MIC and could thus be the best choice for performing energy lookup in MC codes.

References

[1] F.X. Hugot, E. Brun, F. Malvagi, J.C. Trama, and T. Visonneau. High Performance Monte Carlo Computing with TRIPOLI®: Present and Future. In *ANS M&C2015*, LaGrange Park, IL, 2015.

[2] ISO/IEC 14882:2014 – Information technology – Programming languages – C++. 2014.

[3] Forrest B Brown. New hash-based energy lookup algorithm for monte carlo codes. *Trans. Am. Nucl. Soc.*, 111:659–662, 2014.

[4] J. Leppänen. Two practical methods for unionized energy grid construction in continuous-energy Monte Carlo neutron transport calculation. *Annals of Nuclear Energy*, 36(7):878–885, 2009.

[5] A.L. Lund and A.R. Siegel. Using fractional cascading to accelerate cross section lookups in Monte Carlo neutron transport calculations. In *ANS M&C2015*, LaGrange Park, IL, 2015.

[6] J.A. Walsh et al. Optimizations of the energy grid search algorithm in continuous-energy Monte Carlo particle transport codes. *Computer Physics Communications*, 196:134–142, 2015.

[7] P.K. Romano et al. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Annals of Nuclear Energy*, 82:90–97, 2015.

[8] Intel Corporation. *Intel Xeon Phi Coprocessor System Software Developers Guide*, sku 328207-003en edition, March 2014.

[9] John R Tramm and Andrew R Siegel. Memory bottlenecks and memory contention in multi-core monte carlo transport codes. *Annals of Nuclear Energy*, 82:195–202, 2015.

[10] M. Probst. *Linear vs Binary Search*, 2010. https://schani.wordpress.com/tag/c-optimization-linear-binary-search-sse2-simd.